Chapter 7

Input and Output

In This Chapter

- ▶ Using standard input and output
- ▶ Reading and writing characters
- ▶ Understanding `getchar()` and `putchar()`
- ▶ Exploring the `char` variable type
- ▶ Reading input with `scanf()`
- ▶ Grabbing strings with `fgets()`

One of the basic functions, if not *the* basic function, of any computing device is input and output. The old I/O (say "eye oh") is also the goal of just about every program. Input is received and processed, and then output is generated. The processing is what makes the program useful. Otherwise, you'd have only input and output, which is essentially the same thing as plumbing.

Character I/O

The simplest type of input and output takes place at the character level: One character goes in; one character comes out. Of course, getting to that point involves a wee bit of programming.

Understanding input and output devices

The C language was born with the Unix operating system. As such, it follows many of the rules for that operating system with regard to input and output. Those rules are pretty solid:

- Input comes from the standard input device, `stdin`.
- Output is generated by the standard output device, `stdout`.

On a computer, the standard input device, `stdin`, is the keyboard. Input can also be redirected by the operating system, so it can come from another device, like a modem, or it can also come from a file.

The standard output device, `stdout`, is the display. Output can be redirected so that it goes to another device, such as a printer or into a file.

REMEMBER

C language functions that deal with input and output access the `stdin` and `stdout` devices. They do not directly read from the keyboard or output to the screen. Well, unless you code your program to do so. (Such coding isn't covered in this book.)

Bottom line: Although your programs can get input from the keyboard and send output to the screen, you need to think about C language I/O in terms of `stdin` and `stdout` devices instead. If you forget that, you can get into trouble, which I happily demonstrate later in this chapter.

Fetching characters with getchar()

It's time for your code to become more interactive. Consider the source code from Listing 7-1, which uses the `getchar()` function. This function gets (reads) a character from standard input.

Listing 7-1: It Eats Characters

```
#include <stdio.h>

int main()
{
    int c;

    printf("I'm waiting for a character: ");
    c = getchar();
    printf("I waited for the '%c' character.\n",c);
    return(0);
}
```

The code in Listing 7-1 reads a character from standard input by using the `getchar()` function at Line 8. The character is returned from `getchar()` and stored in the *c* integer variable.

Line 9 displays the character stored in *c*. The `printf()` function uses the `%c` placeholder to display single characters.

Exercise 7-1: Type the source code for project ex0701 , as shown in Listing 7-1. Build and run.

The `getchar()` function is defined this way:

```
#include <stdio.h>

int getchar(void);
```

The function has no arguments, so the parentheses are always empty; the word `void` in this example basically says so. And the `getchar()` function requires the `stdio.h` header file to be included with your source code.

`getchar()` returns an integer value, not a `char` variable. The compiler warns you when you forget. And don't worry: The `int` contains a character value, which is just how it works.

Exercise 7-2: Edit Line 9 in the source code from Listing 7-1 so that the `%d` placeholder is used instead of `%c`. Build and run.

The value that's displayed when you run the solution to Exercise 7-1 is the character's ASCII code value. The `%d` displays that value instead of a character value because internally the computer treats all information as values. Only when information is displayed as a character does it look like text.

Appendix A lists ASCII code values.

Technically, `getchar()` is not a function. It's a *macro* — a shortcut based on another function, as defined in the `stdio.h` header file. The real function to get characters from standard input is `getc()`; specifically, when used like this:

```
c = getc(stdin);
```

In this example, `getc()` reads from the standard input device, `stdin`, which is defined in the `stdio.h` header file. The function returns an integer value, which is stored in variable *c*.

Exercise 7-3: Rewrite the source code for Listing 7-1, replacing the `getchar()` statement with the `getc()` example just shown.

Exercise 7-4: Write a program that prompts for three characters; for example:

```
I'm waiting for three characters:
```

Code three consecutive `getchar()` functions to read the characters. Format the result like this:

```
The three characters are 'a', 'b', and 'c'
```

where these characters — `a`, `b`, and `c` — would be replaced by the program's input.

TIP

The program you create in Exercise 7-4 waits for three characters. The Enter key is a character, so if you type **A**, **Enter**, **B**, **Enter**, the three characters are *A*, the Enter key character, and then *B*. That's valid input, but what you probably want to type is something like **ABC** or **PIE** or **LOL** and then press the Enter key.

REMEMBER

Standard input is stream-oriented. As I mention earlier in this chapter, don't expect your C programs to be interactive. Exercise 7-4 is an example of how stream input works; the Enter key doesn't end stream input; it merely rides along in the stream, like any other character.

Using the putchar() function

The evil twin of the `getchar()` function is the `putchar()` function. It serves the purpose of kicking a single character out to standard output. Here's the format:

```
#include <stdio.h>

int putchar(int c);
```

To make `putchar()` work, you send it a character, placing a literal character in the parentheses, as in

```
putchar('v');
```

Or you can place the ASCII code value (an integer) for the character in the parentheses. The function returns the value of the character that's written. (See Listing 7-2.)

Listing 7-2: Putting putchar() to Work

```
#include <stdio.h>

int main()
{
    int ch;

    printf("Press Enter: ");
    getchar();
    ch = 'H';
    putchar(ch);
    ch = 'i';
    putchar(ch);
    putchar('!');
    return(0);
}
```

This chunk of code uses the `getchar()` function to pause the program. The statement in Line 8 waits for input. The input that's received isn't stored; it doesn't need to be. The compiler doesn't complain if you don't keep the value returned from the `getchar()` function (or from any function).

In Lines 9 through 12, the `putchar()` function displays the value of variable *`ch`* one character at a time.

Single-character values are assigned to the *`ch`* variable in Lines 9 and 11. The assignment works just like assigning values, though single characters are specified, enclosed in single quotes. This process still works, even though *`ch`* isn't a `char` variable type.

In Line 13, `putchar()` displays a constant value directly. Again, the character must be enclosed in single quotes.

Exercise 7-5: Create a new project, ex0705, using the source code shown in Listing 7-2. Build and run the program.

One weird thing about the output is that the final character isn't followed by a newline. That output can look awkward on a text display, so:

Exercise 7-6: Modify the source code from Exercise 7-5 so that one additional character is output after the ! character (the newline character).

Working with character variables

The `getchar()` and `putchar()` functions work with integers, but that doesn't mean you need to shun the character variable. The `char` is still a variable type in C. When you work with characters, you use the `char` variable type to store them, as shown in Listing 7-3.

Listing 7-3: Character Variable Madness

```
#include <stdio.h>

int main()
{
    char a,b,c,d;

    a = 'W';
    b = a + 24;
    c = b + 8;
    d = '\n';
    printf("%c%c%c%c",a,b,c,d);
    return(0);
}
```

Exercise 7-7: Create a new project, ex0707, using the source code in Listing 7-3. Build and run the program.

The code declares four `char` variables at Line 5. These variables are assigned values in Lines 7 through 10. Line 7 is pretty straightforward. Line 8 uses math to set the value of variable *`b`* to a specific character, as does Line 9 for variable *`c`*. (Use Appendix A to look up a character's ASCII code value.) Line 10 uses an escape sequence to set a character's value, something you can't type at the keyboard.

All those `%c` placeholders are stuffed into the `printf()` statement, but the output is, well, surprising.

Exercise 7-8: Modify the code for Listing 7-3 so that variables *`b`* and *`c`* are assigned their character values directly using character constants held in single quotes.

Exercise 7-9: Modify the source code again so that `putchar()`, not `printf()`, is used to generate output.

Text I/O, but Mostly I

When character I/O is taken up a notch, it becomes text I/O. The primary text output functions are `puts()` and `printf()`. Other ways to spew text in C exist, but those two functions are the biggies. On the I side of I/O are text input functions. The biggies there are `scanf()` and `fgets()`.

Storing strings

When a program needs text input, it's necessary to create a place to store that text. Right away, you'll probably say, "Golly! That would be a string variable." If you answered that way, I admire your thinking. You're relying upon your knowledge that *text* in C programming is referred to as a *string*.

Alas, you're wrong.

C lacks a string variable type. It does, however, have character variables. Queue up enough of them and you have a string. Or, to put it in programming lingo, you have an *array* of character variables.

The array is a big topic, covered in Chapter 12. For now, be open-minded about arrays and strings and soak in the goodness of Listing 7-4.

Listing 7-4: Stuffing a String into a char Array

```
#include <stdio.h>

int main()
{
    char prompt[] = "Press Enter to explode:";

    printf("%s",prompt);
    getchar();
    return(0);
}
```

Line 5 creates an array of `char` variables. The *array* is a gizmo that lists a bunch of variables all in a row. The `char` array variable is named *`prompt`*, which is immediately followed by empty square brackets. It's the Big Clue that the variable is an array. The array is assigned, via the equal sign, the text enclosed in double quotes.

The `printf()` statement in Line 7 displays the string stored in the `prompt` array. The `%s` conversion character represents the string.

In Line 8, `getchar()` pauses the program, anticipating the Enter key press. The program doesn't follow through by exploding anything, a task I leave up to you to code at a future date.

Exercise 7-10: Create a new project, ex0710, and type the source code from Listing 7-4. Build and run the code.

Exercise 7-11: Modify the source code from Listing 7-4 so that a single string variable holds two lines of text; for example:

```
Program to Destroy the World
Press Enter to explode:
```

Hint: Refer to Table 4-1, in Chapter 4.

REMEMBER

- A string variable in C is really a character array.
- You can assign a value to a `char` array when it's created, similarly to the way you initialize any variable when it's created. The format looks like this:

  ```
  char string[] = "text";
  ```

  In the preceding line, `string` is the name of the `char` array, and `text` is the string assigned to that array.

- You can assign a value to a string, or `char` array, only when it's declared in the code. You cannot reassign or change that value later by using a direct statement, such as

  ```
  prompt = "This is just wrong.";
  ```

  Changing a string is possible in C, but you need to know more about arrays, string functions, and especially pointers before you make the attempt. Later chapters in this book cover those topics.
- See Chapter 6 for an introduction to the basic C language variable types. The full list of C language variables is found in Appendix D.

Introducing the scanf() function

For the input of specific types of variables, you'll find that the `scanf()` function comes in handy. It's not a general-purpose input function, and it has some limitations, but it's great for testing code or grabbing values.

In a way, you could argue that `scanf()` is the input version of the `printf()` function. For example, it uses the same conversion characters (the `%` placeholder-things). Because of that, `scanf()` is quite particular about how text is input. Here's the format:

```
#include <stdio.h>

int scanf(const char *restrict format,...);
```

Scary, huh? Just ignore it for now. Here's a less frightening version of the format:

```
scanf("placeholder",variable);
```

In this version, *`placeholder`* is a conversion character, and *`variable`* is a type of variable that matches the conversion character. Unless it's a string (`char` array), the variable is prefixed by the `&` operator.

The `scanf()` function is prototyped in the `stdio.h` header file, so you must include that file when you use the function.

Here are some `scanf()` examples:

```
scanf("%d",&highscore);
```

The preceding statement reads an integer value into the variable *`highscore`*. I'm assuming that *`highscore`* is an `int` variable.

```
scanf("%f",&temperature);
```

The preceding `scanf()` statement waits for a floating-point value to be input, which is then stored in the *`temperature`* variable.

```
scanf("%c",&key);
```

In the preceding line, `scanf()` accepts the first character input and stores it in the *`key`* variable.

```
scanf("%s",firstname);
```

REMEMBER

The `%s` placeholder is used to read in text, but only until the first white space character is encountered. So a space or a tab or the Enter key terminates the string. (That sucks.) Also, *`firstname`* is a `char` array, so it doesn't need the `&` operator in the `scanf()` function.

Reading a string with scanf()

One of the most common ways to put the `scanf()` function to use is to read in a chunk of text from standard input. To meet that end, the `%s` conversion character is used — just like in `printf()`, but with input instead of output. (See Listing 7-5.)

Listing 7-5: scanf() Swallows a String

```
#include <stdio.h>

int main()
{
    char firstname[15];

    printf("Type your first name: ");
    scanf("%s",firstname);
    printf("Pleased to meet you, %s.\n",firstname);
    return(0);
}
```

Exercise 7-12: Type the source code from Listing 7-5 into a new project, ex0712, in Code::Blocks. Build and run.

Line 5 declares a `char` array — a string variable — named *`firstname`*. The number in the brackets indicates the size of the array, or the total number of characters that can be stored there. The array isn't assigned a value, so it's created empty. Basically, the statement at Line 5 sets aside storage for up to 15 characters.

The `scanf()` function in Line 8 reads a string from standard input and stores it in the *`firstname`* array. The `%s` conversion character directs `scanf()` to look for a string as input, just as `%s` is a placeholder for strings in `printf()`'s output.

Exercise 7-13: Modify the source code from Listing 7-5 so that a second string is declared for the person's last name. Prompt the user for their last name as well, and then display both names by using a single `printf()` function.

- The number in the brackets (refer to Line 5 in Listing 7-5) gives the size of the `char` array, or the length of the string, plus *one*.
- When you create a `char` array, or string variable, ensure that you create it of a size large enough to hold the text. That size should be the maximum number of characters plus *one*.
- The reason for increasing the `char` array size by one is that all strings in C end with a specific termination character. It's the NULL character, which is written as `\0`. The compiler automatically adds the `\0` to the end of string values you create in your source code, as well as text read by various text-input functions. You must remember to add room for that character when you set aside storage for string input.

Reading values with scanf()

The `scanf()` function can do more than read strings. It can read in any value specified by a conversion character, as demonstrated in Listing 7-6.

Listing 7-6: scanf() Eats an Integer

```
#include <stdio.h>

int main()
{
    int fav;

    printf("What is your favorite number: ");
    scanf("%d",&fav);
    printf("%d is my favorite number, too!\n",fav);
    return(0);
}
```

In Listing 7-6, the `scanf()` function reads in an integer value. The `%d` conversion character is used, just like `printf()` — indeed, it's used in Line 9. That character directs `scanf()` to look for an `int` value for variable *`fav`*.

Exercise 7-14: Create a project, ex0714, using the source code shown in Listing 7-6. Build and run. Test the program by typing various integer values, positive and negative.

Perhaps you're wondering about the ampersand (`&`) in the `scanf()` function. The character is a C operator — specifically, the *memory address* operator. It's one of the advanced features in C that's related to pointers. I avoid the topic of pointers until Chapter 18, but for now, know that an ampersand must prefix any variable specified in the `scanf()` function. The exception is an array, such as the `firstname char` array in Listing 7-5.

Try running the program again, but specify a decimal value, such as 41.9, or type text instead of a number.

The reason you see incorrect output is that `scanf()` is *very* specific. It fetches only the variable type specified by the conversion character. So if you want a floating-point value, you must specify a `float` variable and use the appropriate conversion character; `%f`, in that case.

Exercise 7-15: Modify the source code from Listing 7-6 so that a floating-point number is requested, input, and displayed.

REMEMBER

- You don't need to prefix a `char` array variable with an ampersand in the `scanf()` function; when using `scanf()` to read in a string, just specify the string variable name.
- The `scanf()` function stops reading text input at the first white space character, space, tab, or Enter key.

Using fgets() for text input

For a general-purpose text input function, one that reads beyond the first white space character, I recommend the `fgets()` function. Here's the format:

```
#include <stdio.h>

char * fgets(char *restrict s, int n, FILE *restrict
          stream);
```

Frightening, no? That's because `fgets()` is a file function, which reads text from a file, as in "file get string." That's how programmers talk after an all-nighter.

File functions are covered in Chapter 22, but because the operating system considers standard input like a file, you can use `fgets()` to read text from the keyboard.

Here's a simplified version of the `fgets()` function as it applies to reading text input:

```
fgets(string,size,stdin);
```

In this example, *`string`* is the name of a `char` array, a string variable; *`size`* is the amount of text to input plus *one*, which should be the same size as the `char` array; and `stdin` is the name of the standard input device, as defined in the `stdio.h` header file. (See Listing 7-7.)

Listing 7-7: The fgets() Function Reads a String

```
#include <stdio.h>

int main()
{
    char name[10];

    printf("Who are you? ");
    fgets(name,10,stdin);
    printf("Glad to meet you, %s.\n",name);
    return(0);
}
```

Exercise 7-16: Type the source code from Listing 7-7 into a new project, ex0716. Compile and run.

The `fgets()` function in Line 8 reads in text. The text goes into the `name` array, which is set to a maximum of ten characters in Line 5. The number `10` specifies that `fgets()` reads in only nine characters, one less than the number specified. Finally, `stdin` is specified as the "file" from which input is read. `stdin` is *st*and*ard* *in*put.

REMEMBER

The `char` array must have one extra character reserved for the `\0` at the end of a string. Its size must equal the size of input you need — plus one.

Here's how the program runs on my screen:

```
Who are you? Danny Gookin
Glad to meet you, Danny Goo.
```

Only the first nine characters of the text I typed in the first line are displayed. Why only nine? Because of the string's terminating character — the NULL, or `\0`. The room for this character is defined when the `name` array is created in Line 5. If `fgets()` were to read in ten characters instead of nine, the array would overflow, and the program could malfunction.

WARNING!

Avoid `fgets()`'s evil sibling `gets()`

It may seem bizarre to use `fgets()`, a file function, to read in text, but it's perfectly legitimate. The real reason, however, is that the original C language text-input function shouldn't be used at all. That function is the `gets()` function, which is still a valid function in C, but one you should avoid.

Unlike `fgets()`, the `gets()` function reads an infinite amount of text from standard input. It's not limited to a given number of characters, like `fgets()`. A user could input as much text as possible, and the program would continue to store the text, eventually overwriting something else in memory. This weakness has been exploited historically by various types of malware.

Bottom line: Don't use `gets()` — use `fgets()` instead. In fact, when you use `gets()`, you see a warning when you compile your code. The program you create may also display a warning.

Exercise 7-17: Change the array size in the source code from Listing 7-7 to a constant value. Set the constant to allow only three characters input.

Exercise 7-18: Redo your solution for Exercise 7-13 so that `fgets()` rather than `scanf()` is used to read in the two strings.

You can read more about the reason that input is limited by `fgets()` in the nearby sidebar, "The `fgets()` function's evil sibling, `gets()`."

REMEMBER

- The `fgets()` function reads text from standard input, not from the keyboard directly.
- The value returned by `fgets()` is the string that was input. In this book's sample code, that return value isn't used, although it's identical to the information stored in the `fgets()` function's first argument, the `char` array variable.
- Chapter 13 offers more information about strings in C.